

Programowo-Sprzętowa Realizacja Algorytmów:
wyznaczanie i śledzenie punktów
charakterystycznych

Wojciech Gumuła, Rafał Prusak

2 kwietnia 2016

Spis treści

1	Metody wykrywania punktów charakterystycznych.	2
1.1	Metoda Harrisa	3
1.1.1	Implementacja w OpenCV	4
1.2	Metoda Susan	5
1.3	Metoda Fast	6
1.3.1	Implementacja w OpenCV.	7
2	Analiza istniejącego rozwiązania w	9
3	s. śledzenia punktów.	10
4	Opis całości systemu (detekcja + śledzenie z ramki na ramkę).	11
5	Implementacja C++.	12
6	Podział HW/SW.	13
7	Uruchomienie na Zybo.	14

Rozdział 1

Metody wykrywania punktów charakterystycznych.

Ważnym zagadnieniem w systemach wizyjnych jest pozyskiwanie informacji z obrazu, a następnie podejmowanie dalszych działań na podstawie pozyskanych danych.

Wyznaczanie punktów charakterystycznych ma na celu wybór „interesujących” elementów bądź rejonów obrazu oraz uzyskanie charakterystyki specyficznej dla danej aplikacji. Jest to często niskopoziomowa składowa bardziej złożonych algorytmów. Uzyskanie punktów charakterystycznych pozwala na rozpoznawanie kształtów, wyszukiwanie elementów pasujących do wzorca czy też śledzenie obiektów w obrazie wideo.

Istnieje kilka rodzajów punktów unawanych jako charakterystycznych:

- krawędzie,
- narożniki,
- krople(blobs ?).

Szczególnym „powodzeniem” ciszą się narożniki. Punkt te powstają w miejscu krzyżowania się lub gwałtownej zmiany kształtu krawędzi. Z tego powodu, w punkcie narożnym dochodzi do nagłej zmiany wartości gradientu obrazu, co znacząco ułatwia poszukiwanie tego punktu.

1.1 Metoda Harrisa

Algorytm Harrisa polega na analizie jasności obrazu i wyszukiwaniu zmian gradientu.

Poszukiwane są odchylenia jasności:

$$E(u, v) = \sum_{x,y} w(x, y) [I(x + u, y + v) - I(x, y)]^2$$

gdzie:

- $w(x, y)$ - okno w punkcie (x, y) o wymiarach $u \times v$,
- $I(x, y)$ - jasność w punkcie (x, y) ,
- $I(x + u, y + v)$ - jasność w drugim punkcie okna.

Poszukujemy znaczącego odchylenia, więc celem jest maksymalizacja:

$$\sum_{x,y} [I(x + u, y + v) - I(x, y)]^2$$

Po rozwinięciu w szereg Taylora:

$$E(u, v) \approx \sum_{x,y} [I(x, y) + uI_x + vI_y - I(x, y)]^2$$

Dokonujemy skrócenia:

$$E(u, v) \approx \sum_{x,y} u^2 I_x^2 + 2uv I_x I_y + v^2 I_y^2$$

Równanie można zapisać w postaci macierzowej:

$$E(u, v) \approx [u \ v] M \begin{bmatrix} u \\ v \end{bmatrix}$$

z macierzą w postaci:

$$M = \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

Dla każdego okna wyliczana jest wartość wyrażenia:

$$R = \det(M) - k(\text{trace}(M))^2$$

gdzie:

- $\det(M) = \lambda_1 \lambda_2$
- $\text{trace}(M) = \lambda_1 + \lambda_2$

Wartość wyrażenia R powyżej pewnej wartości granicznej oznacza występowanie narożnika.

1.1.1 Implementacja w OpenCV

Poniższy kod prezentuje wykrywanie narożników za pomocą biblioteki OpenCV:

```

1  #include "opencv2/highgui.hpp"
2  #include "opencv2/imgproc.hpp"
3  #include <iostream>
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  using namespace cv;
8  using namespace std;
9
10 Mat src, src_gray;
11 int thresh = 200;
12 int max_thresh = 255;
13
14 char* source_window = "Source_image";
15 char* corners_window = "Corners_detected";
16
17 void cornerHarris_demo( int, void* ) {
18     Mat dst, dst_norm, dst_norm_scaled;
19     dst = Mat::zeros(src.size(), CV_32FC1);
20
21     int blockSize = 2;
22     int apertureSize = 3;
23     double k = 0.04;
24
25     cornerHarris(src_gray, dst, blockSize, apertureSize, k, BORDER_DEFAULT);
26
27     normalize(dst, dst_norm, 0, 255, NORM_MINMAX, CV_32FC1, Mat());
28     convertScaleAbs(dst_norm, dst_norm_scaled);
29
30     for (int j=0; j<dst_norm.rows; j++) {
31         for (int i=0; i<dst_norm.cols; i++) {
32             if((int)dst_norm.at<float>(j,i) > thresh)
33                 circle(dst_norm_scaled, Point(i, j), 5, Scalar(0), 2, 8, 0);
34         }
35     }
36
37     namedWindow(corners_window, WINDOW_AUTOSIZE);
38     imshow(corners_window, dst_norm_scaled);
39 }
40
41 int main( int argc, char** argv ) {
42     src = imread(argv[1], 1);
43     cvtColor(src, src_gray, COLOR_BGR2GRAY);
44
45     namedWindow(source_window, WINDOW_AUTOSIZE);
46     createTrackbar("Threshold:", source_window, &thresh, max_thresh, cornerHarris_demo);
47     imshow(source_window, src);
48     cornerHarris_demo(0, 0);
49     waitKey(0);
50     return(0);
51 }

```



Rysunek 1.1: Wykrywanie narożników metodą Harisa w OpenCV

1.2 Metoda Susan

Algorytm SUSAN (ang. Smallest Univalve Segment Assimilating Nucleus) polega na znajdowaniu pikseli, które w otoczeniu mają najniższą liczbę punktów o zbliżonej jasności.

$$c(r, r_0) = \begin{cases} 1 & \text{gdy } |I(r) - I(r_0)| \leq t \\ 0 & \text{gdy } |I(r) - I(r_0)| > t \end{cases}$$

gdzie:

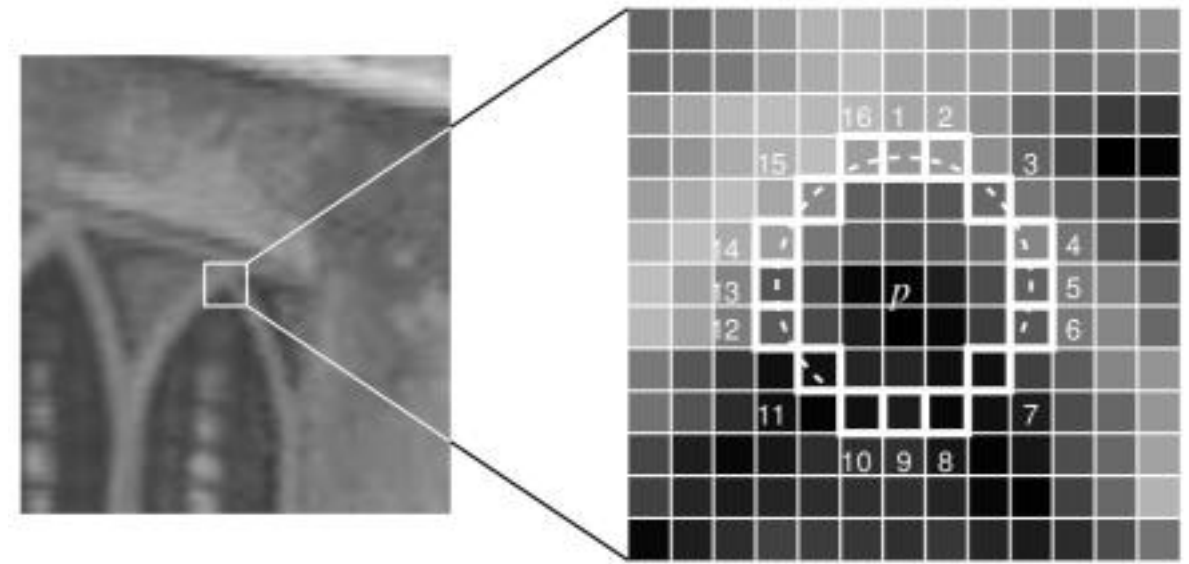
- I - jasność obrazu,
- r_0 - piksel centralny,
- t - próg podobieństwa pikseli

Liczbę pikseli podobnych wyznaczamy następująco:

$$D(r_0) = \sum_{r \in N(r_0)} c(r, r_0)$$

gdzie $N(r_0)$ to otoczenie piksela centralnego.

Aby punkt był uznany za narożnik, co najwyżej połowa punktów jego otoczeniu może być do niego podobna.



Rysunek 1.2: Wybór otoczenia piksela w metodzie FAST.

1.3 Metoda Fast

Algorytm Fast (ang. Features from Accelerated Segment Test) opiera się, podobnie jak Susan, na przeszukaniu otoczenia piksela.

Opis algorytmu:

1. Wybieramy piksel p o poziomie jasności I_p .
2. Wybieramy wartość progu t .
3. Wybieramy okrąg o promieniu $r = 16$ pikseli.
4. Piksel p jest narożnikiem jeżeli na okręgu znajduje się ciąg n pikseli jaśniejszych niż $I_p + t$ lub ciemniejszych niż $I_p - t$.

Algorytm ten został udoskonalony przez dodanie testu, który pozwala bardzo szybko odrzucić dużą liczbę punktów nie będących narożnikami. W tym teście sprawdzana jest jasność po kolei pikseli 1, 9, 5, 13.

Dobór okręgu został przedstawiony na rysunku 1.2.

1.3.1 Implementacja w OpenCV.

Poniższy kod prezentuje wykrywanie narożników metodą Fast za pomocą biblioteki OpenCV:

```

1  #include <opencv2/highgui.hpp>
2  #include <opencv2/imgproc.hpp>
3  #include <opencv2/features2d.hpp>
4  #include <vector>
5  #include <iostream>
6
7  using namespace cv;
8  using namespace std;
9
10 Mat src, src_gray;
11 int thresh = 20;
12 int max_thresh = 50;
13
14 char* source_window = "Source_image";
15 char* corners_window = "Corners_detected";
16
17 void Fast_demo(int, void *) {
18     std::vector<KeyPoint> points;
19     FAST(src_gray, points, thresh);
20
21     auto temp = src_gray.clone();
22
23     for(auto keyPoint: points) {
24         int x = keyPoint.pt.x;
25         int y = keyPoint.pt.y;
26         Point p{x, y};
27         circle(temp, p, 5, Scalar(0));
28     }
29
30     namedWindow(corners_window, WINDOW_AUTOSIZE);
31     imshow(corners_window, temp);
32 }
33
34 int main(int argc, char** argv) {
35     src = imread(argv[1], 1);
36     cvtColor(src, src_gray, COLOR_BGR2GRAY);
37     namedWindow(source_window, WINDOW_AUTOSIZE);
38     createTrackbar("Threshold: ", source_window, &thresh, max_thresh, Fast_demo);
39     imshow(source_window, src);
40     Fast_demo(0, 0);
41     waitKey(0);
42     return(0);
43 }

```




Rysunek 1.3: Wykrywanie punktów charakterystycznych metodą Fast ($t=50$).

Rozdział 2

Analiza istniejącego rozwiązania w

Rozdział 3

s. śledzenia punktów.

Rozdział 4

Opis całości systemu
(detekcja + śledzenie z ramki
na ramkę).

Rozdział 5

Implementacja C++.

Rozdział 6

Podział HW/SW.

Rozdział 7

Uruchomienie na Zybo.